PUCRS ESCOLA POLITÉCNICA

Algoritmos de Divisão e Conquista: Quicksort

O ALGORITMO:

- Proposto por C.A.R Hoare em 1962.
- No Quicksort o processo de dividir e conquistar ocorre em três etapas para ordenar um vetor de tamanho $\mathrm{A}[p{:}\,r]$:
- **Dividir:** particionando (reorganizando) o vetor $\mathrm{A}[p{:}\,r]$ em dois subvetores $\mathrm{A}[p{:}\,q-1]$(o lado inferior) e $\mathrm{A}[q+1{:}\,r]$(o lado superior) de modo que cada elemento no lado inferior da partição seja menor ou igual ao pivô $\mathrm{A}[q]$, que é, em por sua vez, menor ou igual a cada elemento no lado alto. Calcule o índice q do pivô como parte deste procedimento de particionamento.
- **Conquistar:** chamando $QUICKSOR$ recursivamente para ordenar cada um dos subvetores.
- **Combinar** sem fazer nada: como os dois subvetores já estão ordenados, não é necessário combiná-los. Todos os elementos em $\mathrm{A}[p{:}\,q-1]$ estão ordenados e são menores ou iguais a $\mathrm{A}[q]$ e todos os elementos em $\mathrm{A}[q+1{:}\,r]$ estão ordenados e são maiores ou iguais a $\mathrm{A}[q]$.

- O procedimento $QUICKSORT$ implementa quicksort. Para ordenar um vetor $\mathrm{A}[1{:}\,n]$, a chamada inicial será $QUICKSORT(A, 1, n)$.

```
QUICKSORT(A, p, r)
if p < r
    // Partition the subarray around the pivot, which ends up in A[q].
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)  // recursively sort the low side
    QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

Particionando o Vetor:

- A chave do algoritmo é o procedimento $PARTITION$, que reorganiza o subvetor no lugar, retornando o índice do ponto de divisão entre os dois lados da partição.

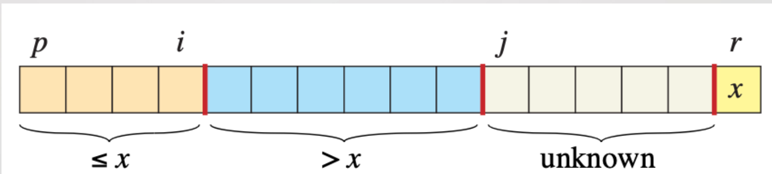
```
PARTITION(A, p, r)
x = A[r]                          // the pivot
i = p − 1                         // highest index into the low side
for j = p to r − 1                // process each element other than the pivot
    if A[j] ≤ x                   // does this element belong on the low side?
        i = i + 1                 // index of a new slot in the low side
        exchange A[i] with A[j]   // put this element there
exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
return i + 1                      // new index of the pivot
```

Algoritmos de Divisão e Conquista: Quicksort

Particionando o Vetor:

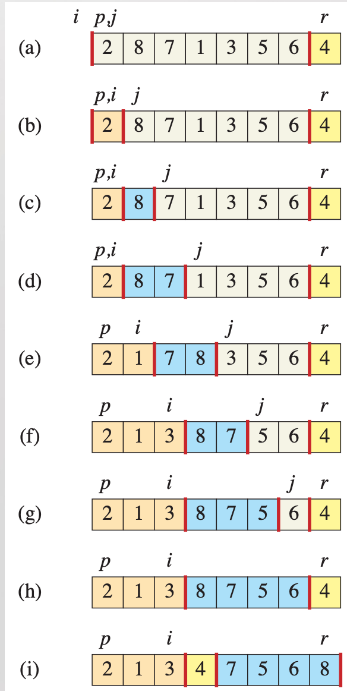
```
PARTITION(A, p, r)
1  x = A[r]                              // the pivot
2  i = p − 1                             // highest index into the low side
3  for j = p to r − 1                    // process each element other than the pivot
4      if A[j] ≤ x                       // does this element belong on the low side?
5          i = i + 1                     // index of a new slot in the low side
6          exchange A[i] with A[j]       // put this element there
7  exchange A[i + 1] with A[r]           // pivot goes just to the right of the low side
8  return i + 1                          // new index of the pivot
```

- A Figura mostra como $PARTITION$ funciona em um vetor de $n = 8$ elementos.
- PARTITION sempre seleciona o elemento $\mathrm{x} = \mathrm{A}[r]$ como pivô.
- À medida que o procedimento é executado, cada elemento cai exatamente em uma das quatro regiões, algumas das quais podem estar vazias.



- No início de cada iteração do loop for nas linhas 3-6, as regiões satisfazem certas propriedades, mostradas. Declaramos essas propriedades como um loop invariante.
- No início de cada iteração do loop de linhas 3-6, para qualquer índice k, as seguintes condições são válidas e invariantes durante o looping:

1. Se $p \leq k \leq i$, então $\mathrm{A}[k] \leq x$ (região laranja)
2. Se $i + 1 \leq k \leq j - 1$, então $\mathrm{A}[k] > x$ (região azul).
3. Se $k = r$, então $\mathrm{A}[k] = x$ (região amarela).



ANALISANDO O ALGORITMO:

```
PARTITION(A, p, r)
1  x = A[r]                           // the pivot
2  i = p - 1                          // highest index into the low side
3  for j = p to r - 1                 // process each element other than the pivot
4      if A[j] ≤ x                    // does this element belong on the low side?
5          i = i + 1                  // index of a new slot in the low side
6          exchange A[i] with A[j]    // put this element there
7  exchange A[i + 1] with A[r]        // pivot goes just to the right of the low side
8  return i + 1                       // new index of the pivot
```

- No início de cada iteração do loop de linhas 3-6, para qualquer índice k, as seguintes condições são válidas e invariantes durante o looping:
    1. Se $p \leq k \leq i$, então $\mathrm{A}[k] \leq x$ (região laranja)
    2. Se $i + 1 \leq k \leq j - 1$, então $\mathrm{A}[k] > x$ (região azul).
    3. Se $k = r$, então $\mathrm{A}[k] = x$ (região amarela).
- Precisamos mostrar que o loop invariante de $PARTITION$ é:
    - ❖ verdadeiro antes da primeira iteração,
    - ❖ que cada iteração do loop mantém o invariante,
    - ❖ que o loop termina e
    - ❖ que a correção segue do invariante quando o loop termina.

- **Inicialização:** Antes da primeira iteração do loop, temos $i = p - 1$ e $j = p$.

  Como nenhum valor está entre p e i e nenhum valor está entre $i + 1$ e $j - 1$, as duas primeiras condições do invariante do loop são trivialmente satisfeitas.

  A atribuição na linha 1 ($x = \mathrm{A}[r]$) satisfaz a terceira condição.

Algoritmos de Divisão e Conquista: Quicksort

ANALISANDO O ALGORITMO:

```
PARTITION(A, p, r)
1  x = A[r]                        // the pivot
2  i = p − 1                       // highest index into the low side
3  for j = p to r − 1              // process each element other than the pivot
4      if A[j] ≤ x                 // does this element belong on the low side?
5          i = i + 1               // index of a new slot in the low side
6          exchange A[i] with A[j] // put this element there
7  exchange A[i + 1] with A[r]     // pivot goes just to the right of the low side
8  return i + 1                    // new index of the pivot
```

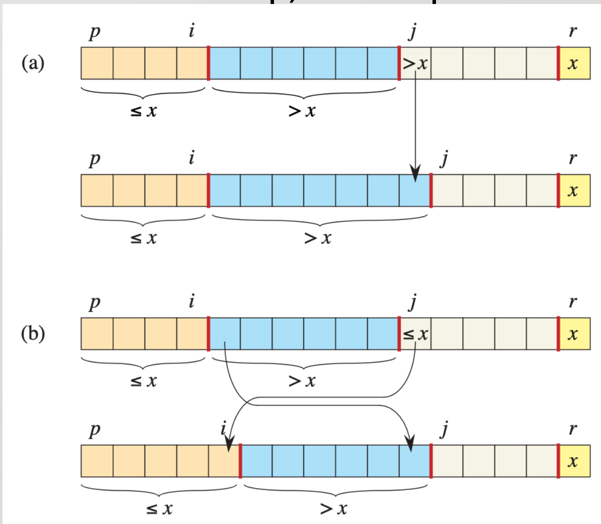
- No início de cada iteração do loop de linhas 3-6, para qualquer índice k, as seguintes condições são válidas e invariantes durante o looping:
    1. Se $p \leq k \leq i$, então $\mathrm{A}[k] \leq x$ (região laranja)
    2. Se $i + 1 \leq k \leq j - 1$, então $\mathrm{A}[k] > x$ (região azul).
    3. Se $k = r$, então $\mathrm{A}[k] = x$ (região amarela).
- Precisamos mostrar que o loop invariante de $PARTITION$ é:
    - ❖ verdadeiro antes da primeira iteração,
    - ❖ que cada iteração do loop mantém o invariante,
    - ❖ que o loop termina e
    - ❖ que a correção segue do invariante quando o loop termina.

- **Manutenção:** Como mostra a Figura, consideramos dois casos, dependendo do resultado do teste na linha 4.

  A Figura (a) mostra o que acontece quando $\mathrm{A}[j] > x$: a única ação no loop é incrementar j. Depois de incrementado j, a segunda condição é válida para $\mathrm{A}[j-1]$ e todas as outras entradas permanecem inalteradas.

  A Figura (b) mostra o que acontece quando $\mathrm{A}[j] \leq x$: o loop aumenta i, troca $\mathrm{A}[i]$ e $\mathrm{A}[j]$, e então aumenta j. Por causa da troca, agora temos $\mathrm{A}[i] \leq x$ e a condição 1 foi satisfeita.

  Da mesma forma, também temos que $\mathrm{A}[j-1] > x$, já que o item para o qual foi trocado para $\mathrm{A}[j-1]$ é, pela invariante do loop, maior que x.



PUCRS ESCOLA POLITÉCNICA

Algoritmos de Divisão e Conquista: Quicksort

ANALISANDO O ALGORITMO:

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p - 1                         // highest index into the low side
3  for j = p to r - 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                 // index of a new slot in the low side
6          exchange A[i] with A[j]   // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

- No início de cada iteração do loop de linhas 3-6, para qualquer índice k, as seguintes condições são válidas e invariantes durante o looping:
  1. Se $p \leq k \leq i$, então $A[k] \leq x$ (região laranja)
  2. Se $i + 1 \leq k \leq j - 1$, então $A[k] > x$ (região azul).
  3. Se $k = r$, então $A[k] = x$ (região amarela).
- Precisamos mostrar que o loop invariante de $PARTITION$ é:
  - ❖ verdadeiro antes da primeira iteração,
  - ❖ que cada iteração do loop mantém o invariante,
  - ❖ que o loop termina e
  - ❖ que a correção segue do invariante quando o loop termina.

- **Finalização:** Como o loop faz exatamente $r - p$ iterações, ele termina quando $j = r$.

  Nesse ponto, o subvetor não examinado $A[j\!:r - 1]$ está vazio e cada entrada no vetor pertence a um dos outros três conjuntos descritos pelo invariante.

  Assim, os valores no vetor foram particionados em três conjuntos: aqueles menores ou iguais a x (o lado inferior), aqueles maiores que x (o lado superior) e um conjunto singleton contendo x (o pivô).
- As duas últimas linhas de $PARTITION$ terminam trocando o pivô pelo esquerdo com o elemento mais a esquerda maior que x, movendo assim o pivô para seu lugar correto no vetor particionado.
- Em seguida, retornando o novo índice do pivô.
- A saída de $PARTITION$ agora atende às especificações fornecidas para a etapa de divisão.
- Na verdade, satisfaz uma condição um pouco mais forte: após a linha 3 do $QUICKSORT$, $A[q]$ é estritamente menor que todos os elementos de $A[q + 1\!:r]$.

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q - 1)   // recursively sort the low side
5      QUICKSORT(A, q + 1, r)   // recursively sort the high side
```

PERFORMANCE E TEMPO DE EXECUÇÃO:

```
QUICKSORT(A, p, r)
if p < r
    // Partition the subarray around the pivot, which ends up in A[q].
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)   // recursively sort the low side
    QUICKSORT(A, q + 1, r)   // recursively sort the high side
```

```
PARTITION(A, p, r)
x = A[r]                             // the pivot
i = p − 1                            // highest index into the low side
for j = p to r − 1                   // process each element other than the pivot
    if A[j] ≤ x                      // does this element belong on the low side?
        i = i + 1                        // index of a new slot in the low side
        exchange A[i] with A[j]          // put this element there
exchange A[i + 1] with A[r]          // pivot goes just to the right of the low side
return i + 1                         // new index of the pivot
```

- O tempo de execução do $QUICKSORT$ depende de quão balanceado é cada particionamento, que por sua vez depende de quais elementos são usados como pivôs.
- Se os dois lados de uma partição tiverem aproximadamente o mesmo tamanho, o particionamento será balanceado, então o algoritmo será executado assintoticamente tão rápido quanto a ordenação por $MERGESORT$.
- Se o particionamento estiver desequilibrado, entretanto, ele poderá ser executado assintoticamente tão lentamente quanto a ordenação por $INSERTIONSORT$.

Algoritmos de Divisão e Conquista: Quicksort

PERFORMANCE E TEMPO DE EXECUÇÃO:

```
QUICKSORT(A, p, r)
if p < r
    // Partition the subarray around the pivot, which ends up in A[q].
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)  // recursively sort the low side
    QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

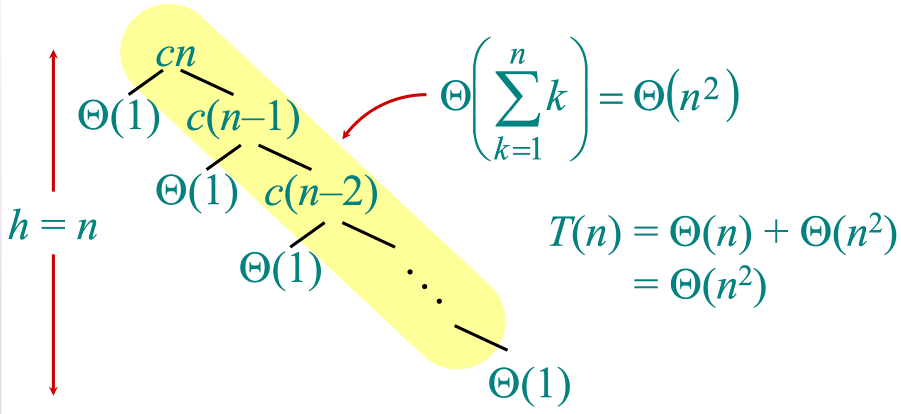
```
PARTITION(A, p, r)
x = A[r]                        // the pivot
i = p − 1                       // highest index into the low side
for j = p to r − 1              // process each element other than the pivot
    if A[j] ≤ x                 // does this element belong on the low side?
        i = i + 1               // index of a new slot in the low side
        exchange A[i] with A[j] // put this element there
exchange A[i + 1] with A[r]     // pivot goes just to the right of the low side
return i + 1                    // new index of the pivot
```

Pior caso de particionamento (worst case)

- O pior comportamento do $QUICKSORT$ ocorre quando o particionamento produz um subproblema com $n - 1$ elementos e outro com 0 elementos.
- Suponhamos que esse particionamento desequilibrado surja em cada chamada recursiva.
- O particionamento custa tempo $\Theta(n)$.
- Como a chamada recursiva em um vetor de tamanho 0 retorna sem fazer nada, $T(0) = \Theta(1)$, a recorrência para o tempo de execução é

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

- A árvore de recursão fica:



- Ao somar os custos incorridos em cada nível da recursão, obtemos uma série aritmética, que resulta em $\Theta(n^2)$.
- Assim, se o particionamento for desequilibrado ao máximo em todos os níveis recursivos do algoritmo, o tempo de execução será $\Theta(n^2)$.

PERFORMANCE E TEMPO DE EXECUÇÃO:

```
QUICKSORT(A, p, r)
if p < r
    // Partition the subarray around the pivot, which ends up in A[q].
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)   // recursively sort the low side
    QUICKSORT(A, q + 1, r)   // recursively sort the high side
```

```
PARTITION(A, p, r)
x = A[r]                          // the pivot
i = p − 1                         // highest index into the low side
for j = p to r − 1                // process each element other than the pivot
    if A[j] ≤ x                   // does this element belong on the low side?
        i = i + 1                 // index of a new slot in the low side
        exchange A[i] with A[j]   // put this element there
exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
return i + 1                      // new index of the pivot
```

Melhor caso de particionamento (best case)

- Na divisão mais uniforme possível, $PARTITION$ produz dois subproblemas, cada um com tamanho não superior a $n/2$.
- Nesse caso, o $QUICKSORT$ é executado muito mais rápido.

- Um limite superior no tempo de execução pode então ser descrito pela recorrência

$$T(n) = 2T(n/2) + \Theta(n)$$

- Pelo caso 2 do teorema mestre, esta recorrência tem a solução $\Theta(n \log_2 n)$.
- Assim, se o particionamento for igualmente equilibrado em todos os níveis da recursão, resultará um algoritmo assintoticamente mais rápido.

Algoritmos de Divisão e Conquista: Quicksort

PERFORMANCE E TEMPO DE EXECUÇÃO:

```
QUICKSORT(A, p, r)
  if p < r
      // Partition the subarray around the pivot, which ends up in A[q].
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q − 1)   // recursively sort the low side
      QUICKSORT(A, q + 1, r)   // recursively sort the high side
```

```
PARTITION(A, p, r)
  x = A[r]                          // the pivot
  i = p − 1                         // highest index into the low side
  for j = p to r − 1                // process each element other than the pivot
      if A[j] ≤ x                   // does this element belong on the low side?
          i = i + 1                 // index of a new slot in the low side
          exchange A[i] with A[j]   // put this element there
  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
  return i + 1                      // new index of the pivot
```

Particionamento Balanceado

- O tempo médio de execução do $QUICKSORT$ está muito mais próximo do melhor caso do que do pior caso.
- Ao apreciar como o equilíbrio do particionamento afeta a recorrência que descreve o tempo de execução, podemos entender o porquê.

- Suponha, por exemplo, que o algoritmo de particionamento sempre produza uma divisão proporcional, o que à primeira vista parece bastante desequilibrado. Obtemos então a recorrência:

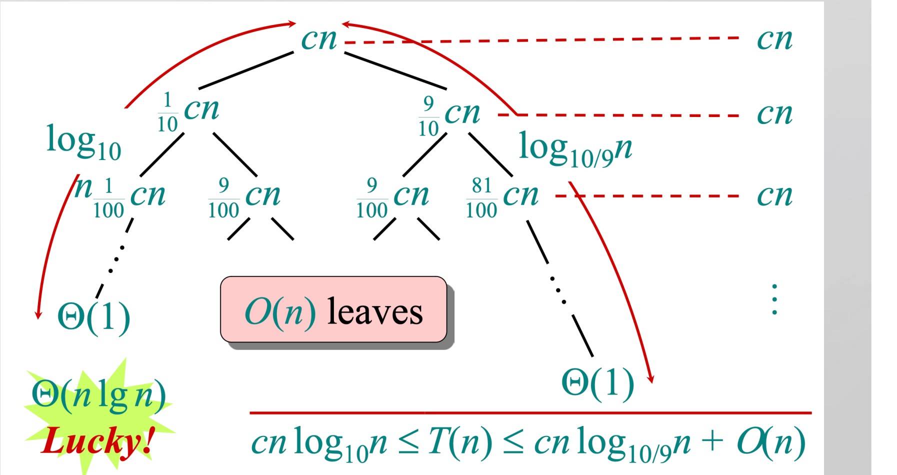
$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + \Theta(n)$$

PERFORMANCE E TEMPO DE EXECUÇÃO:

$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + \Theta(n)$$

- A Figura mostra a árvore de recursão para esta recorrência, onde por simplicidade a função de divisão $\Theta(n)$ foi substituída por cn, o que não afetará a solução assintótica da recorrência.
- Podemos observar dois caminhos extremos na árvore o da esquerda que resulta na aplicação recorrente de $T\left(\frac{1}{10}n\right)$ que chega no caso base com menos recursões; e o direita que resulta na aplicação recorrente de $T\left(\frac{9}{10}n\right)$ e que chega no caso base com mais recursões;
- Cada nível da árvore tem custo cn, até que a recursão da esquerda chegue ao fundo num caso base na profundidade $\log_{10} n = \Theta(\lg n)$, e então os níveis têm custo de no máximo cn.
- A recursão da direita que termina numa profundidade $\log_{10/9} n = \Theta(\lg n)$.

- Assim, com uma divisão 9:1 proporcional em todos os níveis de recursão, o que intuitivamente parece altamente desequilibrado, o $QUICKSORT$ é executado em tempo $O(n \lg n)$, assintoticamente da mesma forma como se a divisão fosse feita bem no meio.
- Na verdade, mesmo uma divisão 99:1 produz um tempo de execução $O(n \lg n)$.
- De fato, qualquer divisão de proporcionalidade constante produz uma árvore de recursão de profundidade $\Theta(\lg n)$, onde o custo em cada nível é $\Theta(n)$.
- O tempo de execução é, portanto, $O(n \lg n)$ sempre que a divisão tiver proporcionalidade constante. A proporção da divisão afeta apenas a constante oculta na notação O.



Algoritmos de Divisão e Conquista: Quicksort

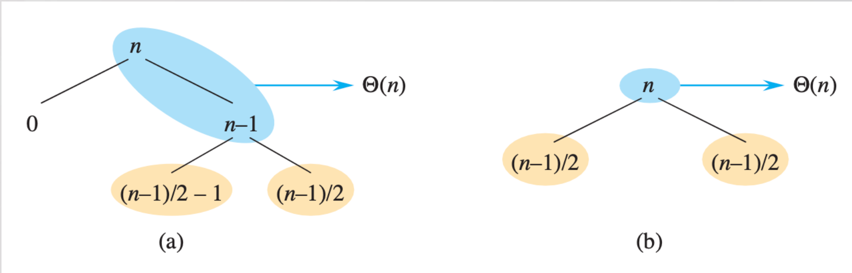
PERFORMANCE E TEMPO DE EXECUÇÃO:

Particionamento Mediano

- Para desenvolver uma noção clara do comportamento esperado do $QUICKSORT$, devemos assumir algo sobre como suas entradas são distribuídas.
- Como o $QUICKSORT$ determina a ordem de classificação usando apenas comparações entre elementos de entrada, seu comportamento depende da ordem relativa dos valores nos elementos do vetor fornecidos como entrada, e não dos valores específicos do vetor.
- Quando o $QUICKSORT$ é executado em um vetor de entrada aleatória, é altamente improvável que o particionamento aconteça da mesma maneira em todos os níveis, como nossa análise informal assumiu.
- Esperamos que algumas das divisões sejam razoavelmente bem equilibradas e que outras sejam bastante desequilibradas.
- No caso médio, $PARTITION$ produz uma mistura de divisões boas e ruins.
- Em uma árvore de recursão para uma execução de caso médio de PARTITION, as divisões boas e ruins são distribuídas aleatoriamente por toda a árvore.
- Suponhamos, por uma questão de intuição, que as divisões boas e más alternam níveis na árvore, e que as divisões boas são divisões de melhor caso e as divisões ruins são divisões de pior caso. A Figura (a) mostra as divisões em dois níveis consecutivos na árvore de recursão :

$$T^{worstCase}(n) = T(n-1) + T(0) + \Theta(n)$$

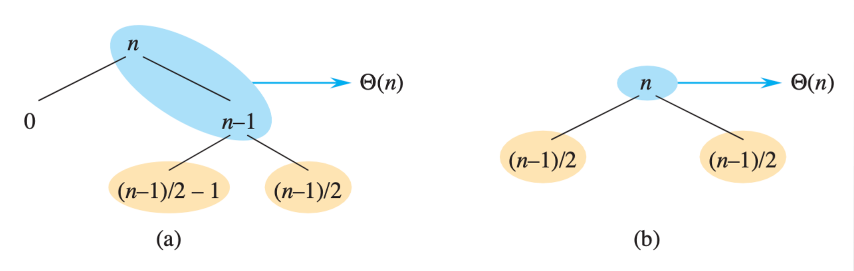
$$T^{bestCase}T(n) = 2T(n/2) + \Theta(n)$$



Algoritmos de Divisão e Conquista: Quicksort

PERFORMANCE E TEMPO DE EXECUÇÃO:

Particionamento Mediano



- Na raiz da árvore, o custo é n para particionamento, e os sub-vetores produzidos têm tamanhos 0 e $n-1$: o pior caso.
- No próximo nível, o sub-vetor de tamanho $n-1$ sofre particionamento no melhor caso em submatrizes de tamanho $(n-1)/2-1$ e $(n-1)/2$.
- Vamos supor que o custo do caso base seja 1 para ao sub-vetor de tamanho 0.
- A combinação da divisão ruim seguida pela divisão boa produz três sub-vetores de tamanhos 0, $(n-1)/2-1$ e $(n-1)/2$ a um custo de particionamento combinado de $\Theta(n)+\Theta(n-1)=\Theta(n)$.
- Esta situação é, no máximo, um fator constante pior do que a da Figura(b), ou seja, onde um único nível de particionamento produz dois sub-vetores de tamanho $(n-1)/2$, a um custo de $\Theta(n)$.
- No entanto, esta última situação é equilibrada!
- Intuitivamente, o custo da divisão ruim na Figura(a) pode ser absorvido pelo custo da divisão boa, e a divisão resultante é boa.
- Assim, o tempo de execução do $QUICKSORT$, quando os níveis alternam entre divisões boas e ruins, é como o tempo de execução apenas para divisões boas: ainda $O(n \lg n)$, mas com uma constante um pouco maior escondida pela notação O.
- Portanto, na média o $QUICKSORT$ é um dos melhores métodos de ordenação de vetores.